ECMASCRIPT 6

PRZEWODNIK PO NOWYM
STANDARDZIE JĘZYKA JAVASCRIPT

NICHOLAS C. ZAKAS

Tytuł oryginału: Understanding ECMAScript 6: The Definitive Guide for JavaScript Developers

Tłumaczenie: Robert Górczyński

ISBN: 978-83-283-3403-8

Title of English-language original: Understanding ECMAScript 6,
ISBN 978-1-59327-757-4, published by No Starch Press.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/ecmasc
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

9

Wprowadzenie do klas JavaScript

W przeciwieństwie do większości formalnych języków programowania zorientowanego obiektowo, JavaScript nie obsługuje klas i ich dziedziczenia jako podstawowego sposobu definiowania podobnych i powiązanych ze sobą obiektów w chwili ich tworzenia. Takie podejście wprowadziło sporo zamieszania wśród programistów. Od specyfikacji w wersji sprzed ECMAScript 1 do ECMAScript 5 wiele bibliotek dostarczało narzędzia, dzięki którym mogło się wydawać, że JavaScript obsługuje klasy. Wprawdzie pewna część programistów była mocno przekonana, że język nie potrzebuje klas, ale ilość utworzonych bibliotek przeznaczonych do obsługi klas spowodowała dołączenie klas w specyfikacji ECMAScript 6.

Podczas poznawania klas w specyfikacji ECMAScript 6 pomocne będzie zrozumienie sposobu działania mechanizmu używanego przez klasy. Dlatego też na początek pokażę, jak programiści uzyskiwali w ECMAScript 5 zachowanie przypominające klasy. Jednak — jak się przekonasz — klasy wprowadzone w specyfikacji ECMAScript 6 nie przypominają klas znanych z innych języków programowania. Ich unikatowość wynika z dynamicznej natury języka JavaScript.

Przypominające klasy struktury w ECMAScript 5

Jak wcześniej wspomniałem, w specyfikacji ECMAScript 5 i wcześniejszych język JavaScript nie oferował klas. Najbliższym odpowiednikiem klasy było utworzenie konstruktora, a następnie przypisanie metod prototypowi konstruktora. Tego rodzaju podejście było zwykle określane mianem utworzenia własnego typu. Spójrz na poniższy fragment kodu.

```
function PersonType(name) {
    this.name = name;
}

PersonType.prototype.sayName = function() {
    console.log(this.name);
};

var person = new PersonType("Nicholas");
person.sayName();    // Dane wyjściowe to "Nicholas".

console.log(person instanceof PersonType);  // Wartość true.
console.log(person instanceof Object);      // Wartość true.
```

W powyższym fragmencie kodu `PersonType` to funkcja konstruktora tworząca pojedynczą właściwość o nazwie `name`. Metoda `sayName()` jest przypisana prototypowi, aby ta sama funkcja mogła być współdzielona przez wszystkie egzemplarze obiektu `PersonType`. Kolejnym krokiem jest utworzenie nowego egzemplarza `PersonType` za pomocą operatora `new`. Otrzymany w wyniku obiekt `person` jest uznawany za egzemplarz `PersonType` i `Object` poprzez dziedziczenie prototypowe.

Tego rodzaju podstawowy wzorzec jest stosowany w wielu bibliotekach JavaScript próbujących zapewnić obsługę klas w języku. To miejsce jest początkiem klas w specyfikacji ECMAScript 6.

Deklaracja klasy

Najprostszą postacią klasy w ECMAScript 6 jest deklaracja klasy, która przedstawia się podobnie do klas używanych w innych językach programowania.

Podstawowa deklaracja klasy

Deklaracja klasy rozpoczyna się od słowa kluczowego `class`, po którym znajduje się nazwa klasy. Pozostała część składni przypomina zwięzłe metody w literale obiektu, choć między elementami klasy nie są wymagane przecinki. Poniżej przedstawiłem przykład prostej deklaracji klasy.

```
class PersonClass {

    // Odpowiednik konstruktora PersonType.
    constructor(name) {
        this.name = name;
    }

    // Odpowiednik PersonType.prototype.sayName.
    sayName() {
        console.log(this.name);
    }
}

let person = new PersonClass("Nicholas");
person.sayName();    // Dane wyjściowe to "Nicholas".

console.log(person instanceof PersonClass);          // Wartość true.
console.log(person instanceof Object);               // Wartość true.

console.log(typeof PersonClass);                     // "function".
console.log(typeof PersonClass.prototype.sayName);   // "function".
```

Deklaracja klasy dla `PersonClass` zachowuje się podobnie jak `PersonType` w poprzednim przykładzie. Jednak zamiast zdefiniować funkcję jako konstruktora, deklaracja klasy pozwala na zdefiniowanie konstruktora bezpośrednio w klasie za pomocą metody specjalnej o nazwie `constructor`. Ponieważ metody klasy używają zwięzłej składni, więc nie ma potrzeby stosowania słowa kluczowego `function`. Nazwy wszystkich pozostałych metod nie mają specjalnego znaczenia, możesz dodać dowolną liczbę metod.

Własne właściwości to właściwości egzemplarza, a nie prototypu, i dlatego mogą być tworzone tylko wewnątrz metody lub konstruktora klasy. W omawianym przykładzie `name` jest własną właściwością. Zalecam, aby w funkcji konstruktora utworzyć wszystkie możliwe własne właściwości. W ten sposób mamy w klasie pojedyncze miejsce odpowiedzialne za te wszystkie właściwości.

Interesujący może być fakt, że deklaracje klasy to po prostu syntaktyczny dodatek dla istniejących deklaracji typów niestandardowych. Deklaracja `Person➥Class` tworzy funkcję posiadającą zachowanie metody `constructor` i dlatego też wartością zwrotną wywołania `typeof PersonClass` jest `"function"`. Metoda `sayName()` w omawianym przykładzie również kończy jako metoda w `PersonClass.➥prototype`, podobnie jak w przypadku związku między `sayName()` i `PersonType.➥prototype` w poprzednich przykładach. Te podobieństwa pozwalają na łączenie niestandardowych typów i klas bez większych obaw o to, które z nich są używane.

UWAGA *Prototypy klasy, takie jak `PersonClass.prototype` w poprzednim przykładzie, przeznaczone są tylko do odczytu. Oznacza to brak możliwości przypisania nowej wartości prototypowi, co można zrobić w przypadku funkcji.*

Dlaczego należy używać składni klasy?

Pomimo pewnych podobieństw między klasami i typami niestandardowymi, należy pamiętać o istnieniu ważnych różnic między nimi.

- W przeciwieństwie do deklaracji funkcji, deklaracje klas nie podlegają hoistingowi. Deklaracja klasy działa w charakterze deklaracji `let`, więc istnieje w tymczasowo martwej strefie aż do chwili, gdy wykonywanie programu dotrze do tej deklaracji.
- Cały kod znajdujący się w deklaracji klasy jest automatycznie uruchamiany w trybie ścisłym. Nie ma żadnej możliwości rezygnacji z trybu ścisłego wewnątrz klasy.
- Żadna z metod nie poddaje się wyliczeniu. To istotna zmiana w stosunku do typu niestandardowego, w którym trzeba użyć wywołania `Object.defineProperty()`, aby metoda nie poddawała się wyliczeniu.
- Żadna metoda nie ma wewnętrznej metody `[[Construct]]` i zgłosi błąd, jeśli spróbujesz wywoływać ją wraz ze słowem kluczowym `new`.
- Wywołanie konstruktora klasy bez słowa kluczowego `new` powoduje zgłoszenie błędu.
- Próba nadpisania nazwy klasy metodą klasy powoduje zgłoszenie błędu.

Mając na uwadze te wszystkie różnice, możemy zobaczyć, że przedstawiona w poprzednim przykładzie deklaracja `PersonClass` jest bezpośrednim odpowiednikiem poniższego kodu, w którym nie użyliśmy składni klasy.

```
// Bezpośredni odpowiednik PersonClass.
let PersonType2 = (function() {

    "use strict";

    const PersonType2 = function(name) {

        // Upewniamy się, że funkcja została wywołana wraz ze słowem kluczowym new.
        if (typeof new.target === "undefined") {
            throw new Error("Konstruktor musi być wywołany wraz ze słowem
            ↪kluczowym new.");
        }

        this.name = name;
    }

    Object.defineProperty(PersonType2.prototype, "sayName", {
        value: function() {

            // Upewniamy się, że metoda nie została wywołana wraz ze słowem
            // kluczowym new.
            if (typeof new.target !== "undefined") {
```

```
                throw new Error("Metoda nie może być wywołana wraz ze
                ↪słowem kluczowym new.");
            }

            console.log(this.name);
        },
        enumerable: false,
        writable: true,
        configurable: true
    });

    return PersonType2;
}());
```

Przede wszystkim zwróć uwagę na istnienie dwóch deklaracji `PersonType2`. Pierwsza to deklaracja `let` w zewnętrznym zasięgu, natomiast druga to deklaracja `const` w natychmiast wykonywanym wyrażeniu funkcji. Teraz już wiesz, dlaczego metody klasy nie mogą nadpisywać nazwy klasy, choć może to zrobić kod zdefiniowany poza klasą. Funkcja konstruktora sprawdza `new.target`, aby mieć pewność wywołania wraz ze słowem kluczowym `new`. W przeciwnym razie nastąpi zgłoszenie błędu. Następnie metoda `sayName()` zostaje zdefiniowana jako niepoddająca się wyliczeniu i sprawdza `new.target`, aby mieć pewność, że nie była wywołana wraz z `new`. Ostatnim krokiem jest zwrot funkcji konstruktora.

W omówionym przykładzie pokazałem, że można wykorzystać możliwości klas bez użycia nowej składni, ale ta właśnie składnia znacznie upraszcza funkcjonalność.

NAZWY STAŁYCH KLASY

Nazwa klasy to jedyna stała w klasie. Oznacza to, że można nadpisać nazwę klasy na zewnątrz niej, ale nie w metodzie klasy. Spójrz na poniższy fragment kodu.

```
class Foo {
   constructor() {
       Foo = "bar";   // Zgłoszenie błędu podczas wykonywania kodu…
   }
}

// Takie podejście jest dozwolone, ale po deklaracji klasy.
Foo = "baz";
```

W powyższym fragmencie kodu `Foo` wewnątrz konstruktora klasy to wiązanie oddzielne od `Foo` na zewnątrz klasy. Wewnętrzne `Foo` zostało zdefiniowane jako stała, której nie można nadpisać. Kiedy konstruktor spróbuje nadpisać to `Foo` dowolną wartością, zostanie zgłoszony błąd. Z kolei zewnętrzne `Foo` zostało zdefiniowane za pomocą deklaracji `let` i dlatego w każdym momencie można nadpisać jego wartość.

Wyrażenia klasy

Klasy i funkcje są podobne pod tym względem, że mają dwie formy — deklaracje i wyrażenia. Deklaracja funkcji i klasy rozpoczyna się od właściwego słowa kluczowego (odpowiednio — `function` i `class`), po którym znajduje się identyfikator. Funkcje mają formę wyrażenia niewymagającą podawania identyfikatora po słowie kluczowym `function`. Podobnie klasy mają formę wyrażenia niewymagającą podawania identyfikatora po słowie kluczowym `class`. Te **wyrażenia klasy** zostały zaprojektowane do użycia w deklaracjach zmiennych lub przekazywania funkcjom jako argumenty.

Podstawowe wyrażenie klasy

Poniżej przedstawiłem odpowiednik wyrażenia klasy dla wcześniejszych przykładów `PersonClass`. Następnie znajduje się pewien kod, w którym użyto tych wyrażeń klasy.

```
let PersonClass = class {

    // Odpowiednik konstruktora PersonType.
    constructor(name) {
        this.name = name;
    }

    // Odpowiednik PersonType.prototype.sayName.
    sayName() {
        console.log(this.name);
    }
};

let person = new PersonClass("Nicholas");
person.sayName();    // Dane wyjściowe to "Nicholas".

console.log(person instanceof PersonClass);          // Wartość true.
console.log(person instanceof Object);               // Wartość true.

console.log(typeof PersonClass);                     // "function".
console.log(typeof PersonClass.prototype.sayName);   // "function".
```

Jak możesz zobaczyć w powyższym fragmencie kodu, wyrażenie klasy nie wymaga identyfikatora po słowie kluczowym `class`. Kiedy pominiemy składnię, wyrażenia klasy są pod względem funkcjonalnym odpowiednikami deklaracji klas. W wyrażeniach klas anonimowych, jak w powyższym przykładzie, `PersonClass.name` to pusty ciąg tekstowy. Kiedy używasz deklaracji klasy, wówczas wartością `PersonClass.name` będzie ciąg tekstowy `"PersonClass"`.

To, czy korzystasz z deklaracji klasy, czy z wyrażenia klasy, to przede wszystkim kwestia stylu. W przeciwieństwie do deklaracji i wyrażeń funkcji, deklaracje

i wyrażenia klasy nie są poddawane hoistingowi, więc wybór nie ma większego wpływu na zachowanie kodu podczas jego wykonywania. Jedyna istotna różnica polega na tym, że wyrażenia klas anonimowych mają właściwość `name` o wartości w postaci pustego ciągu tekstowego, natomiast deklaracje klas zawsze mają właściwość `name` o wartości odpowiadającej nazwie klasy (gdy np. używasz deklaracji klasy, wartością `PersonClass.name` jest `"PersonClass"`).

Wyrażenia nazwanych klas

W poprzednim przykładzie używaliśmy wyrażeń klas anonimowych, ale podobnie jak w wyrażeniach funkcji, można również stosować wyrażenia nazwanych klas. W tym celu po słowie kluczowym `class` należy umieścić identyfikator, np. tak jak pokazałem poniżej.

```
let PersonClass = class PersonClass2 {

    // Odpowiednik konstruktora PersonType.
    constructor(name) {
        this.name = name;
    }

    // Odpowiednik PersonType.prototype.sayName.
    sayName() {
        console.log(this.name);
    }
};

console.log(typeof PersonClass);        // "function".
console.log(typeof PersonClass2);       // "undefined".
```

W powyższym przykładzie wyrażenie klasy nosi nazwę `PersonClass2`. Ten identyfikator istnieje tylko w definicji klasy, więc może być używany wewnątrz metod klasy (np. w metodzie `sayName()`). Poza klasą wywołanie `typeof PersonClass2` zwróci wartość `"undefined"`, ponieważ nie istnieje tutaj żadne wiązanie `PersonClass2`. Aby zrozumieć, dlaczego tak się dzieje, spójrz na odpowiadającą poprzedniemu przykładowi deklarację, która nie używa klas.

```
// Bezpośredni odpowiednik wyrażenia nazwanej klasy PersonClass.
let PersonClass = (function() {

    "use strict";

    const PersonClass2 = function(name) {

        // Upewniamy się, że funkcja została wywołana wraz ze słowem kluczowym new.
        if (typeof new.target === "undefined") {
```

```
            throw new Error("Konstruktor musi być wywołany wraz ze słowem
            ↪kluczowym new.");
        }

        this.name = name;
    }

    Object.defineProperty(PersonClass2.prototype, "sayName", {
        value: function() {

            // Upewniamy się, że metoda nie została wywołana wraz ze słowem
            // kluczowym new.
            if (typeof new.target !== "undefined") {
                throw new Error("Metoda nie może być wywołana wraz
                ↪ze słowem kluczowym new.");
            }

            console.log(this.name);
        },
        enumerable: false,
        writable: true,
        configurable: true
    });

    return PersonClass2;
}());
```

Utworzenie wyrażenia nazwanej klasy nieco zmienia to, co się dzieje w silniku JavaScript. W przypadku deklaracji klasy zewnętrzne wiązanie (zdefiniowane za pomocą `let`) ma taką samą nazwę jak wiązanie wewnętrzne (zdefiniowane za pomocą `const`). W wyrażeniu nazwanej klasy zastosowano nazwę z definicji `const` i dlatego wyrażenie `PersonClass2` jest zdefiniowane do użycia tylko wewnątrz klasy.

Wprawdzie wyrażenia nazwanych klas zachowują się nieco inaczej niż wyrażenia nazwanych funkcji, ale między nimi występują także pewne podobieństwa. Oba rodzaje wyrażeń mogą być używane jako wartości, co otwiera wiele możliwości, którymi zajmę się w kolejnym podrozdziale.

Klasa jako typ pierwszoklasowy

W programowaniu **typ pierwszoklasowy** (ang. *first-class citizen*) to wartość, która może być przekazana funkcji, zwrócona przez funkcję i przypisana zmiennej. Funkcje JavaScript są typami pierwszoklasowymi (nazywane są również **funkcjami pierwszoklasowymi**), które jednocześnie stanowią o unikatowości języka JavaScript.

Specyfikacja ECMAScript 6 kontynuuje tę tradycję przez uznanie klasy także za typ pierwszoklasowy, co pozwala na ich użycie na wiele różnych sposobów. Przykładowo klasę można przekazać funkcji jako argument, tak jak pokazałem poniżej.

```
function createObject(classDef) {
    return new classDef();
}

let obj = createObject(class {

    sayHi() {
        console.log("Cześć!");
    }
});

obj.sayHi();          // "Cześć!".
```

W powyższym fragmencie kodu funkcja `createObject()` jest wywoływana wraz z argumentem w postaci wyrażenia klasy anonimowej, tworzy egzemplarz tej klasy za pomocą słowa kluczowego `new` i zwraca ten egzemplarz. Następnie zmienna `obj` przechowuje zwrócony egzemplarz.

Innym przykładem użycia wyrażenia klasy jest utworzenie egzemplarza typu singleton za pomocą natychmiast wywoływanego konstruktora klasa. W tym celu trzeba użyć słowa kluczowego `new` wraz z wyrażeniem klasy i umieścić na końcu nawias. Tego rodzaju rozwiązanie przedstawiłem w poniższym fragmencie kodu.

```
let person = new class {

    constructor(name) {
        this.name = name;
    }

    sayName() {
        console.log(this.name);
    }
}("Nicholas");

person.sayName();     // "Nicholas".
```

W omawianym przykładzie wyrażenie klasy anonimowej zostaje utworzone, a później natychmiast wykonane. Wzorzec ten pozwala na skorzystanie ze składni klasy do tworzenia egzemplarzy typu singleton bez pozostawienia dostępnego odwołania do klasy w celu analizy. Nawias na końcu wyrażenia klasy wskazuje na wywołanie funkcji, a ponadto pozwala na przekazanie argumentu.

Przykłady przedstawione dotąd w rozdziale były skoncentrowane na klasach wraz z metodami. Jednak istnieje również możliwość utworzenia właściwości akcesora w klasach za pomocą składni podobnej do stosowanej w literałach obiektów.

Właściwości akcesora

Wprawdzie powinieneś tworzyć własne właściwości w konstruktorze klasy, ale klasa pozwala też na zdefiniowanie właściwości akcesora w prototypie. W celu utworzenia metody getter użyj słowa kluczowego `get`, spacji i identyfikatora. Natomiast w celu utworzenia metody setter zastosuj takie samo podejście, ale użyj słowa kluczowego `set`. Spójrz na poniższy fragment kodu.

```
class CustomHTMLElement {

    constructor(element) {
        this.element = element;
    }

    get html() {
        return this.element.innerHTML;
    }

    set html(value) {
        this.element.innerHTML = value;
    }
}

var descriptor =
Object.getOwnPropertyDescriptor(CustomHTMLElement.prototype, "html");
console.log("get" in descriptor);           // Wartość true.
console.log("set" in descriptor);           // Wartość true.
console.log(descriptor.enumerable);         // Wartość false.
```

W omawianym przykładzie klasa `CustomHTMLElement` została użyta w charakterze opakowania dla istniejącego elementu modelu DOM. Ma metody getter i setter dla właściwości `html` delegującej do metody `innerHTML` elementu. Ta właściwość akcesora została utworzona w `CustomHTMLElement.prototype` i podobnie jak każda inna metoda, także i ona nie poddaje się wyliczeniu. Poniżej przedstawiłem niezawierający klasy odpowiednik omówionego rozwiązania.

```
// Bezpośredni odpowiednik poprzedniego przykładu.
let CustomHTMLElement = (function() {

    "use strict";

    const CustomHTMLElement = function(element) {
```

```
        // Upewniamy się, że funkcja została wywołana wraz ze słowem kluczowym new.
        if (typeof new.target === "undefined") {
            throw new Error("Konstruktor musi być wywołany wraz ze słowem
            ↪kluczowym new.");
        }

        this.element = element;
    }

    Object.defineProperty(CustomHTMLElement.prototype, "html", {
        enumerable: false,
        configurable: true,
        get: function() {
            return this.element.innerHTML;
        },
        set: function(value) {
            this.element.innerHTML = value;
        }
    });

    return CustomHTMLElement;
}());
```

Przykład ten, podobnie jak poprzednie, także pokazuje, jak można uniknąć konieczności utworzenia sporej ilości kodu przy użyciu klasy, a nie odpowiednika niezawierającego klasy. Definicja samej właściwości akcesora `html` zabiera prawie połowę wielkości odpowiednika deklaracji klasy.

Generowane nazwy elementów składowych

Podobieństw między literałami obiektu i klasami istnieje znacznie więcej. Metody klasy i właściwości akcesora mogą mieć wygenerowane nazwy. Zamiast korzystać z identyfikatora, należy użyć nawiasu kwadratowego zawierającego wyrażenie. Takie podejście odpowiada składni używanej do generowania nazw literałów obiektu. Spójrz na poniższy fragment kodu.

```
let methodName = "sayName";

class PersonClass {

    constructor(name) {
        this.name = name;
    }
    [methodName]() {
        console.log(this.name);
    }
```

```
};

let me = new PersonClass("Nicholas");
me.sayName();             // "Nicholas".
```

Powyższa wersja `PersonClass` używa zmiennej w celu przypisania nazwy metodzie wewnątrz jej definicji. Ciąg tekstowy `"sayName"` zostaje przypisany zmiennej `methodName`, która następnie będzie użyta do zadeklarowania metody. Później dostęp do metody `sayName()` odbywa się bezpośrednio.

Właściwości akcesora mogą dokładnie tak samo używać wygenerowanych nazw. Spójrz na poniższy fragment kodu.

```
let propertyName = "html";

class CustomHTMLElement {

    constructor(element) {
        this.element = element;
    }

    get [propertyName]() {
        return this.element.innerHTML;
    }

    set [propertyName](value) {
        this.element.innerHTML = value;
    }
}
```

W omawianym przykładzie metody getter i setter dla `html` zostały zdefiniowane za pomocą zmiennej `propertyName`. Uzyskanie dostępu do właściwości przy użyciu składni `.html` wpływa jedynie na definicję.

Poznałeś wiele podobieństw między klasami i literałami obiektów, m.in. metody, właściwości akcesora i generowane nazwy. Istnieje jeszcze jedno podobieństwo, którym chciałbym się zająć, czyli generatory.

Metody generatora

Przypomnij sobie z rozdziału 8. o możliwości zdefiniowania generatora w literale obiektu przez poprzedzenie gwiazdką nazwy metody. Ta sama składnia sprawdza się również w przypadku klas i pozwala, aby dowolna metoda stała się generatorem. Poniżej przedstawiłem przykład tego rodzaju rozwiązania.

```
class MyClass {

    *createIterator() {
```

```
        yield 1;
        yield 2;
        yield 3;
    }
}

let instance = new MyClass();
let iterator = instance.createIterator();
```

W powyższym fragmencie kodu powstała klasa o nazwie `MyClass` wraz z metodą generatora `createIterator()`. Metoda zwraca iterator, którego wartości są na stałe zdefiniowane w generatorze. Metody generatorów okazują się użyteczne, gdy mamy obiekt przedstawiający kolekcję wartości, przez które chcemy łatwo przeprowadzić iterację. Dla tablic, zbiorów i map istnieje sporo metod generatorów, co pozwala programistom pracować na wiele różnych sposobów z obiektami wymienionych typów.

Wprawdzie metody generatorów są użyteczne, ale zdefiniowanie domyślnego iteratora dla klasy będzie znacznie bardziej przydatne, jeśli klasa ta przedstawia kolekcję wartości. Aby zdefiniować domyślny iterator dla klasy, można użyć `Symbol.iterator` w celu określenia metody generatora. Spójrz na poniższy fragment kodu.

```
class Collection {

    constructor() {
        this.items = [];
    }

    *[Symbol.iterator]() {
        yield *this.items.values();
    }
}

var collection = new Collection();
collection.items.push(1);
collection.items.push(2);
collection.items.push(3);

for (let x of collection) {
    console.log(x);
}

// Dane wyjściowe:
// 1
// 2
// 3
```

W przykładzie użyto nazwy wygenerowanej dla generatora, który deleguje iterator `values()` do tablicy `this.items`. Każda klasa zarządzająca kolekcją wartości powinna zawierać domyślny iterator, ponieważ pewne operacje typowe dla

kolekcji wymagają jego dostępności. Teraz możesz już dowolnego egzemplarza `Collection` użyć bezpośrednio w pętli `for-of` lub wraz z operatorem rozszczepiania.

Dodanie metod i właściwości akcesora do prototypu klasy jest użyteczne, gdy chcesz je pokazać w egzemplarzach obiektu. Jeżeli z kolei potrzebujesz metod lub właściwości akcesora w klasie, wówczas musisz użyć statycznych elementów składowych.

Statyczne elementy składowe

Dodanie metod bezpośrednio w konstruktorze w celu symulacji statycznych elementów składowych to kolejny wzorzec często stosowany w kodzie zgodnym ze specyfikacją ECMAScript 5 i wcześniejszymi. Spójrz na poniższy fragment kodu.

```
function PersonType(name) {
    this.name = name;
}

// Metoda statyczna.
PersonType.create = function(name) {
    return new PersonType(name);
};

// Metoda egzemplarza.
PersonType.prototype.sayName = function() {
    console.log(this.name);
};

var person = PersonType.create("Nicholas");
```

W innych językach programowania metoda fabryki o nazwie `PersonType.create()` będzie uznawana za metodę statyczną, ponieważ jej dane nie zależą od egzemplarza typu `PersonType`. Klasa w specyfikacji ECMAScript 6 ułatwia tworzenie statycznych elementów składowych przez użycie formalnej adnotacji `static` przed nazwą metody lub właściwości akcesora. Poniżej przedstawiłem oparty na klasie odpowiednik poprzedniego przykładu.

```
class PersonClass {

    // Odpowiednik konstruktora PersonType.
    constructor(name) {
        this.name = name;
    }

    // Odpowiednik PersonType.prototype.sayName.
    sayName() {
        console.log(this.name);
```

```
    }

    // Odpowiednik PersonType.create.
    static create(name) {
        return new PersonClass(name);
    }
}

let person = PersonClass.create("Nicholas");
```

Definicja `PersonClass` ma pojedynczą metodę statyczną o nazwie `create()`. Składnia metody jest taka sama jak składnia użyta dla `sayName()`, z wyjątkiem słowa kluczowego `static`. Istnieje możliwość użycia słowa kluczowego `static` w definicji dowolnej metody lub właściwości akcesora w klasie. Jedyne ograniczenie wiąże się z brakiem możliwości użycia słowa kluczowego `static` w definicji metody `constructor`.

UWAGA *Statyczne elementy składowe są nieodstępne z poziomu egzemplarzy. Dlatego też dostęp do statycznych elementów składowych musi odbywać się bezpośrednio z poziomu klasy.*

Dziedziczenie z użyciem klas pochodnych

Przed wprowadzeniem specyfikacji ECMAScript 6 implementacja dziedziczenia wraz z typami niestandardowymi była dość obszernym procesem. Zastosowanie prawidłowego dziedziczenia wymagało wielu kroków. Spójrz na poniższy fragment kodu.

```
function Rectangle(length, width) {
    this.length = length;
    this.width = width;
}

Rectangle.prototype.getArea = function() {
    return this.length * this.width;
};

function Square(length) {
    Rectangle.call(this, length, length);
}

Square.prototype = Object.create(Rectangle.prototype, {
    constructor: {
        value: Square,
        enumerable: true,
```

```
        writable: true,
        configurable: true
    }
});

var square = new Square(3);

console.log(square.getArea());              // 9.
console.log(square instanceof Square);      // Wartość true.
console.log(square instanceof Rectangle);   // Wartość true.
```

Obiekt `Square` dziedziczy po `Rectangle` i dlatego musi nadpisywać `Square.prototype` nowym obiektem utworzonym przez `Rectangle.prototype`, jak również wywołać metodę `Rectangle.call()`. Kroki te są często niezrozumiałe dla początkujących programistów JavaScript, a także stanowią źródło błędów dla doświadczonych programistów.

Klasy ułatwiają implementację dziedziczenia za pomocą znanego słowa kluczowego `extends` pozwalającego na wskazanie funkcji, po której dana klasa powinna dziedziczyć. Prototypy są automatycznie dostosowywane i można uzyskać dostęp do bazowego konstruktora klasy przez wywołanie metody `super()`. Poniżej przedstawiłem zgodny ze specyfikacją ECMAScript 6 odpowiednik poprzedniego przykładu.

```
class Rectangle {
    constructor(length, width) {
        this.length = length;
        this.width = width;
    }

    getArea() {
        return this.length * this.width;
    }
}

class Square extends Rectangle {
    constructor(length) {

        // Odpowiednik wywołania Rectangle.call(this, length, length).
        super(length, length);
    }
}

var square = new Square(3);

console.log(square.getArea());              // 9.
console.log(square instanceof Square);      // Wartość true.
console.log(square instanceof Rectangle);   // Wartość true.
```

Tym razem klasa `Square` dziedziczy po `Rectangle` za pomocą słowa kluczowego `extends`. Konstruktor `Square` używa `super()` do wywołania konstruktora `Rectangle` wraz z podanymi argumentami. Warto zwrócić uwagę, że w przeciwieństwie do wersji kodu dla specyfikacji ECMAScript 5, identyfikator `Rectangle` jest używany jedynie w deklaracji klasy (po słowie kluczowym `extends`).

Klasy dziedziczące po innych klasach są określane mianem **klas pochodnych**. Klasa pochodna wymaga użycia wywołania `super()`, jeśli jest podawany konstruktor, w przeciwnym razie nastąpi zgłoszenie błędu. Jeżeli zdecydujesz się na nieużywanie konstruktora, metoda `super()` będzie wywoływana automatycznie po utworzeniu nowego egzemplarza klasy. Przykładowo dwie poniższe klasy są identyczne.

```
class Square extends Rectangle {
    // Brak konstruktora.
}

// Powyższa definicja jest odpowiednikiem poniższej.

class Square extends Rectangle {
    constructor(...args) {
        super(...args);
    }
}
```

Druga klasa w omawianym przykładzie pokazuje odpowiednik konstruktora domyślnego dla wszystkich klas pochodnych. Konstruktorowi klasy bazowej są po kolei przekazywane wszystkie argumenty. W omawianym przykładzie funkcjonalność nie jest całkiem poprawna, ponieważ konstruktor `Square` wymaga tylko jednego argumentu i dlatego najlepszym rozwiązaniem będzie ręczne zdefiniowanie konstruktora.

UWAGI DOTYCZĄCE UŻYCIA METODY SUPER()

Podczas stosowania metody `super()` pamiętaj o wymienionych poniżej kwestiach.

- Metody `super()` można używać jedynie w konstruktorze klasy pochodnej. Jeżeli spróbujesz ją wywołać w innej klasie (czyli nieużywającej słowa kluczowego `extends`) lub w funkcji, wówczas nastąpi zgłoszenie błędu.
- Metodę `super()` trzeba wywołać przed uzyskaniem dostępu do wiązania `this` w konstruktorze. Ponieważ metoda jest odpowiedzialna za inicjalizację `this`, więc próba uzyskania dostępu do `this` przed wywołaniem `super()` skutkuje zgłoszeniem błędu.
- Jedynym sposobem na uniknięcie wywołania metody `super()` jest zwrot obiektu przy użyciu konstruktora klasy.

Przesłanianie metod klasy

Metody w klasach pochodnych zawsze przesłaniają metody o takich samych nazwach zdefiniowane w klasie bazowej. Możesz np. dodać metodę `getAdd()` do `Square`, aby przedefiniować tę funkcjonalność.

```
class Square extends Rectangle {
    constructor(length) {
        super(length, length);
    }

    // Nadpisanie i przesłonięcie metody Rectangle.prototype.getArea().
    getArea() {
        return this.length * this.length;
    }
}
```

Ponieważ metoda `getArea()` została zdefiniowana jako część egzemplarza `Square`, więc metoda o nazwie `Rectangle.prototype.getArea()` nie będzie dłużej wywoływana przez jakikolwiek egzemplarz `Square`. Oczywiście zawsze możesz się zdecydować na wywołanie metody w wersji zdefiniowanej w klasie bazowej. W omawianym przypadku oznacza to konieczność użycia wywołania `super.get` ↪`Area()`, tak jak pokazałem poniżej.

```
class Square extends Rectangle {
    constructor(length) {
        super(length, length);
    }

    // Nadpisanie, przesłonięcie i wywołanie metody Rectangle.prototype.getArea().
    getArea() {
        return super.getArea();
    }
}
```

Użycie `super` w przedstawiony sposób działa dokładnie tak samo jak odwołanie `super` omówione w rozdziale 4. Wartość `this` będzie automatycznie prawidłowo ustawiona, aby można było wykonać proste wywołanie metody.

Dziedziczone statyczne elementy składowe

Jeżeli klasa bazowa ma statyczne elementy składowe, będą one dostępne również w klasie pochodnej. Dziedziczenie działa podobnie jak w innych językach programowania, ale jest to nowa koncepcja w JavaScripcie. Spójrz na poniższy fragment kodu.

```
class Rectangle {
    constructor(length, width) {
        this.length = length;
        this.width = width;
    }

    getArea() {
        return this.length * this.width;
    }

    static create(length, width) {
        return new Rectangle(length, width);
    }
}

class Square extends Rectangle {
    constructor(length) {

        // Odpowiednik Rectangle.call(this, length, length).
        super(length, length);
    }
}

var rect = Square.create(3, 4);

console.log(rect instanceof Rectangle);      // Wartość true.
console.log(rect.getArea());                 // 12.
console.log(rect instanceof Square);         // Wartość false.
```

W powyższym fragmencie kodu do klasy `Rectangle` została dodana nowa metoda statyczna o nazwie `create()`. Poprzez mechanizm dziedziczenia metoda ta stała się dostępna jako `Square.create()` i zachowuje się jak metoda `Rectangle.create()`.

Klasy pochodne na podstawie wyrażeń

Prawdopodobnie oferującym największe możliwości aspektem klas pochodnych w specyfikacji ECMAScript 6 jest możliwość utworzenia klasy pochodnej na podstawie wyrażenia. Słowa kluczowego `extends` można użyć wraz z dowolnym wyrażeniem, o ile będzie się ono sprowadzało do funkcji wraz z metodą `[[Construct]]` i prototypem. Spójrz na poniższy fragment kodu.

```
function Rectangle(length, width) {
    this.length = length;
    this.width = width;
}
```

```
Rectangle.prototype.getArea = function() {
    return this.length * this.width;
};

class Square extends Rectangle {
    constructor(length) {
        super(length, length);
    }
}

var x = new Square(3);
console.log(x.getArea());               // 9.
console.log(x instanceof Rectangle);    // Wartość true.
```

Konstruktor `Rectangle` to konstruktor zdefiniowany w stylu ECMAScript 5, natomiast `Square` to klasa. Ponieważ `Rectangle` zawiera metodę `[[Construct]]` i prototyp, więc klasa `Square` nadal może bezpośrednio po nim dziedziczyć.

Akceptacja dowolnego typu wyrażenia po słowie kluczowym `extends` oferuje użyteczne możliwości, takie jak dynamiczne ustalenie, co i po czym należy dziedziczyć. Spójrz na poniższy fragment kodu.

```
function Rectangle(length, width) {
    this.length = length;
    this.width = width;
}

Rectangle.prototype.getArea = function() {
    return this.length * this.width;
};

function getBase() {
    return Rectangle;
}

class Square extends getBase() {
    constructor(length) {
        super(length, length);
    }
}

var x = new Square(3);
console.log(x.getArea());               // 9.
console.log(x instanceof Rectangle);    // Wartość true.
```

Funkcja `getBase()` jest wywoływana bezpośrednio jako część deklaracji klasy. Zwraca egzemplarz `Rectangle` i dlatego pod względem funkcjonalności przykład ten jest odpowiednikiem poprzedniego. Ponieważ klasę bazową można

ustalić dynamicznie, więc stało się możliwe zastosowanie innych podejść podczas dziedziczenia. Można np. utworzyć domieszkę, tak jak pokazałem poniżej.

```
let SerializableMixin = {
    serialize() {
        return JSON.stringify(this);
    }
};

let AreaMixin = {
    getArea() {
        return this.length * this.width;
    }
};

function mixin(...mixins) {
    var base = function() {};
    Object.assign(base.prototype, ...mixins);
    return base;
}

class Square extends mixin(AreaMixin, SerializableMixin) {
    constructor(length) {
        super();
        this.length = length;
        this.width = length;
    }
}

var x = new Square(3);
console.log(x.getArea());                    // 9.
console.log(x.serialize());                  // "{"length":3,"width":3}"
```

W przykładzie użyłem domieszki zamiast klasycznego dziedziczenia. Funkcja `mixin()` pobiera dowolną liczbę argumentów przedstawiających obiekty domieszki. Tworzy funkcję o nazwie `base` i przypisuje prototypowi właściwości poszczególnych obiektów domieszki. Następnie funkcja zostaje zwrócona, aby możliwe było użycie słowa kluczowego `extends` wraz ze `Square`. Pamiętaj, że w konstruktorze wciąż konieczne będzie wywołanie `super()`, ponieważ nadal jest używane słowo kluczowe `extends`.

Egzemplarz `Square` zawiera metody `getArea()` z `AreaMixin` oraz `serialize()` z `SerializableMixin`. Stało się to możliwe za sprawą dziedziczenia prototypowego. Funkcja `mixin()` dynamicznie wypełnia prototyp nową funkcją wraz z własnymi właściwościami poszczególnych domieszek. Warto pamiętać, że jeśli kilka domieszek ma tę samą właściwość, wówczas w nowym obiekcie pozostanie tylko ostatnia z dodanych.

UWAGA *Wprawdzie po słowie kluczowym `extends` można użyć dowolnego wyrażenia, ale nie wszystkie spowodują powstanie prawidłowej klasy. Szczególnie użycie po `extends` wyrażenia zwracającego wartość `null` lub funkcję generatora (omówione w rozdziale 8.) spowoduje zgłoszenie błędu. W takim przypadku próba utworzenia nowego egzemplarza klasy spowoduje zgłoszenie błędu, ponieważ nie będzie metody `[[Construct]]` do wywołania.*

Dziedziczenie po wbudowanych klasach

Niemal od samego początku istnienia tablic w języku JavaScript programiści chcieli mieć możliwość tworzenia własnych specjalnych typów tablic za pomocą dziedziczenia. W specyfikacji ECMAScript 5 i wcześniejszych nie było to możliwe. Próba zastosowania klasycznego dziedziczenia nie skutkowała powstaniem prawidłowo funkcjonującego kodu. Spójrz na poniższy fragment kodu.

```
// Zachowanie wbudowanej tablicy.
var colors = [];
colors[0] = "czerwony";
console.log(colors.length);          // 1.

colors.length = 0;
console.log(colors[0]);              // undefined.

// Próba dziedziczenia po tablicy w specyfikacji ES5.

function MyArray() {
    Array.apply(this, arguments);
}

MyArray.prototype = Object.create(Array.prototype, {
    constructor: {
        value: MyArray,
        writable: true,
        configurable: true,
        enumerable: true
    }
});

var colors = new MyArray();
colors[0] = "czerwony";
console.log(colors.length);          // 0.

colors.length = 0;
console.log(colors[0]);              // "czerwony".
```

Wywołanie `console.log()` na końcu tego fragmentu kodu pokazuje, że użycie w JavaScripcie klasycznej postaci dziedziczenia po tablicy powoduje nieoczekiwane zachowanie. Właściwość `length` i właściwości liczbowe w egzemplarzu

MyArray nie zachowują się w taki sam sposób jak we wbudowanej tablicy, ponieważ ta funkcjonalność nie jest obsługiwana przez Array.apply() lub za pomocą przypisania prototypu.

Jednym z celów klas w specyfikacji ECMAScript 6 było umożliwienie dziedziczenia po wszystkich wbudowanych typach danych. Aby osiągnąć ten cel, model dziedziczenia klas jest pod dwoma ważnymi względami nieco inny niż model dziedziczenia klasycznego znany ze specyfikacji ECMAScript 5 i wcześniejszych.

W przypadku klasycznego dziedziczenia w specyfikacji ECMAScript 5 wartość this jest najpierw tworzona przez typ pochodny (np. MyArray), a następnie mamy wywołanie konstruktora typu bazowego (np. metody Array.apply()). Oznacza to, że this na początku jest egzemplarzem MyArray, a następnie zostanie udekorowane dodatkowymi właściwościami typu Array.

Natomiast w opartym na klasach dziedziczeniu w specyfikacji ECMAScript 6 wartość this jest najpierw tworzona przez typ bazowy (tutaj Array), a dopiero później modyfikowana przez konstruktor klasy pochodnej (tutaj MyArray). Dlatego też this na początku ma całą funkcjonalność wbudowanego typu bazowego i prawidłowo otrzymuje wszelką związaną z nim funkcjonalność.

W poniższym przykładzie pokazałem w działaniu specjalną tablicę opartą na klasie.

```
class MyArray extends Array {
    // Pusta definicja.
}

var colors = new MyArray();
colors[0] = "czerwony";
console.log(colors.length);          // 1.

colors.length = 0;
console.log(colors[0]);              // undefined.
```

Egzemplarz MyArray dziedziczy bezpośrednio po Array i dlatego działa podobnie jak tablica. Użycie właściwości liczbowych powoduje uaktualnienie właściwości length, a zmiana właściwości length pociąga za sobą uaktualnienie właściwości liczbowych. Oznacza to, że można poprawnie dziedziczyć po wbudowanym typie Array, aby utworzyć własne klasy pochodne tablicy, a także prawidłowo dziedziczyć po innych wbudowanych typach danych. Po dodaniu tej funkcjonalności w specyfikacji ECMAScript 6 i klasach pochodnych praktycznie usunięto ostatni przypadek specjalny dziedziczenia po wbudowanych typach danych. Mimo wszystko, nadal warto poznać wspomniany przypadek specjalny.

Właściwość Symbol.species

Wygodnym aspektem dziedziczenia po wbudowanych typach danych jest to, że każda metoda zwracająca egzemplarz wbudowanego typu danych będzie automatycznie zamiast niego zwracała egzemplarz klasy pochodnej. Dlatego też, jeśli

masz klasę pochodną o nazwie `MyArray` dziedziczącą po wbudowanym typie `Array`, metoda, taka jak `slice()`, zwróci egzemplarz typu `MyArray`. Spójrz na poniższy fragment kodu.

```
class MyArray extends Array {
    // Pusta definicja.
}

let items = new MyArray(1, 2, 3, 4),
    subitems = items.slice(1, 3);

console.log(items instanceof MyArray);       // Wartość true.
console.log(subitems instanceof MyArray);    // Wartość true.
```

W powyższym fragmencie kodu metoda `slice()` zwraca egzemplarz `MyArray`. Metoda `slice()` jest dziedziczona po `Array` i normalnie zwraca egzemplarz typu `Array`. Zmiana jest w rzeczywistości przeprowadzana w tle przez właściwość `Symbol.species`.

Właściwość `Symbol.species` jest doskonale znana i używana do zdefiniowania statycznej właściwości akcesora zwracającej funkcję. Ta funkcja będzie konstruktorem przeznaczonym do użycia, gdy egzemplarz klasy ma zostać utworzony w metodzie egzemplarza (zamiast wykorzystania konstruktora). Wymienione poniżej wbudowane typy danych mają zdefiniowaną właściwość `Symbol.species`:

- `Array`;
- `ArrayBuffer` (jego dokładne omówienie znajdziesz w rozdziale 10.);
- `Map`;
- `Promise`;
- `RegExp`;
- `Set`;
- typowane tablice (jego dokładne omówienie znajdziesz w rozdziale 10.).

Wszystkie typy wymienione na powyższej liście mają właściwość `Symbol.` ↪`species` zwracającą wiązanie `this`, co oznacza, że właściwość ta zawsze będzie zwracała funkcję konstruktora. Gdyby tę funkcjonalność zaimplementować w klasie niestandardowej, wówczas kod mógłby przedstawiać się następująco.

```
// Kilka wbudowanych typów danych używa rozwiązania podobnego do poniższego.
class MyClass {
    static get [Symbol.species]() {
        return this;
    }

    constructor(value) {
        this.value = value;
```

```
    }

    clone() {
        return new this.constructor[Symbol.species](this.value);
    }
}
```

W powyższym fragmencie kodu doskonale znany `Symbol.species` został użyty w celu przypisania statycznej właściwości akcesora do `MyClass`. Zwróć uwagę na obecność metody getter bez setter, ponieważ zmiana typu klasy jest niemożliwa. Każde wywołanie `this.constructor[Symbol.species]` zwróci wartość `MyClass`. Metoda `clone()` używa tej definicji w celu zwrotu nowego egzemplarza zamiast bezpośredniego użycia `MyClass`, co spowodowałoby nadpisanie tej wartości przez klasę pochodną. Spójrz na poniższy fragment kodu.

```
class MyClass {
    static get [Symbol.species]() {
        return this;
    }

    constructor(value) {
        this.value = value;
    }
    clone() {
        return new this.constructor[Symbol.species](this.value);
    }
}

class MyDerivedClass1 extends MyClass {
    // Pusta definicja.
}

class MyDerivedClass2 extends MyClass {
    static get [Symbol.species]() {
        return MyClass;
    }
}

let instance1 = new MyDerivedClass1("foo"),
    clone1 = instance1.clone(),
    instance2 = new MyDerivedClass2("bar"),
    clone2 = instance2.clone();

console.log(clone1 instanceof MyClass);              // Wartość true.
console.log(clone1 instanceof MyDerivedClass1);      // Wartość true.
console.log(clone2 instanceof MyClass);              // Wartość true.
console.log(clone2 instanceof MyDerivedClass2);      // Wartość false.
```

W omawianym przykładzie MyDerivedClass1 dziedziczy po MyClass i nie zmienia właściwości Symbol.species. Po wywołaniu metody clone() zwracany jest egzemplarz MyDerivedClass1, ponieważ this.constructor[Symbol.species] zwraca MyDerivedClass1. Klasa MyDerivedClass2 dziedziczy po MyClass i nadpisuje Symbol.species w celu zwrotu MyClass. Po wywołaniu metody clone() w egzemplarzu MyDerivedClass2 wartością zwrotną jest egzemplarz MyClass. Za pomocą Symbol.species i dowolnej klasy pochodnej można ustalić typ wartości, jaka powinna być zwrócona, gdy metoda zwraca egzemplarz.

Przykładowo klasa Array używa Symbol.species do wskazania klasy przeznaczonej do wykorzystania przez metodę zwracającą tablicę. W klasie pochodnej typu Array można ustalić typ obiektu zwracanego przez metody odziedziczone, np. tak jak pokazałem poniżej.

```
class MyArray extends Array {
    static get [Symbol.species]() {
        return Array;
    }
}

let items = new MyArray(1, 2, 3, 4),
    subitems = items.slice(1, 3);

console.log(items instanceof MyArray);          // Wartość true.
console.log(subitems instanceof Array);         // Wartość true.
console.log(subitems instanceof MyArray);       // Wartość false.
```

W powyższym fragmencie kodu nadpisujemy Symbol.species w egzemplarzu MyArray dziedziczącym po Array. Wszystkie metody odziedziczone zwracające tablicę będą teraz używały egzemplarza Array zamiast MyArray.

Ogólnie rzecz biorąc, powinieneś stosować właściwość Symbol.species zawsze wtedy, gdy chcesz użyć this.constructor w metodzie klasy. To pozwala klasom pochodnym na łatwe nadpisanie zwracanego typu. Ponadto jeśli tworzysz klasy pochodne na podstawie klasy zawierającej Symbol.species, upewnij się, że użyłeś tej wartości zamiast konstruktora.

Użycie właściwości new.target w konstruktorze klasy

W rozdziale 3. poznałeś właściwość new.target oraz dowiedziałeś się, jak jej wartość ulega zmianie w zależności od sposobu wywołania funkcji. Istnieje również możliwość użycia właściwości new.target w konstruktorze klasy w celu ustalenia, jak klasa jest wywoływana. W tym prostym przykładzie właściwość new.target jest odpowiednikiem funkcji konstruktora klasy, co pokazałem poniżej.

```
class Rectangle {
    constructor(length, width) {
        console.log(new.target === Rectangle);
        this.length = length;
        this.width = width;
    }
}

// Właściwość new.target ma wartość Rectangle.
var obj = new Rectangle(3, 4);        // Dane wyjściowe to true.
```

W omawianym przykładzie pokazałem, że właściwość `new.target` jest odpowiednikiem `Rectangle` podczas wywołania `new Rectangle(3, 4)`. Konstruktor klasy nie może być wywołany bez `new`, więc właściwość `new.target` zawsze będzie zdefiniowana w konstruktorze klasy. Jednak jej wartość nie zawsze będzie taka sama. Spójrz na poniższy fragment kodu.

```
class Rectangle {
    constructor(length, width) {
        console.log(new.target === Rectangle);
        this.length = length;
        this.width = width;
    }
}

class Square extends Rectangle {
    constructor(length) {
        super(length, length)
    }
}

// Właściwość new.target ma wartość Square.
var obj = new Square(3);        // Dane wyjściowe to false.
```

Egzemplarz `Square` wywołuje konstruktor `Rectangle`, więc podczas wywoływania konstruktora `Rectangle` wartością właściwości `new.target` jest `Square`. To bardzo ważny aspekt, ponieważ daje każdemu konstruktorowi możliwość zmiany zachowania w zależności od sposobu jego wywołania. Przykładowo abstrakcyjną klasę bazową (niemożliwą do bezpośredniego wywołania) można za pomocą właściwości `new.target` zdefiniować w poniższy sposób.

```
// Abstrakcyjna klasa bazowa.
class Shape {
    constructor() {
        if (new.target === Shape) {
            throw new Error("Nie można bezpośrednio utworzyć egzemplarzy
            ↪tej klasy.")
```

```
        }
    }
}

class Rectangle extends Shape {
    constructor(length, width) {
        super();
        this.length = length;
        this.width = width;
    }
}

var x = new Shape();                    // Następuje zgłoszenie błędu.

var y = new Rectangle(3, 4);            // Bez błędu.
console.log(y instanceof Shape);        // Wartość true.
```

W omawianym przykładzie konstruktor klasy `Shape` zgłosi błąd, gdy wartością `new.target` będzie `Shape`, co oznacza, że wywołanie `new Shape()` zawsze zgłosi błąd. Jednak nadal można używać `Shape` jako klasy bazowej i takie podejście zastosowaliśmy w przypadku egzemplarza `Rectangle`. Wywołanie `super()` powoduje wykonanie konstruktora `Shape`, a wartością `new.target` jest `Rectangle`, więc konstruktor bez problemów kontynuuje działanie.

UWAGA *Ponieważ klasa nie może być wywołana bez słowa kluczowego `new`, właściwość `new.target` w konstruktorze klasy nigdy nie będzie miała wartości `undefined`.*

Podsumowanie

Klasy w specyfikacji ECMAScript 6 ułatwiają użycie dziedziczenia w języku JavaScript, a programista może wykorzystać posiadane doświadczenie dotyczące dziedziczenia poznanego w innych językach programowania. Na początku klasy w specyfikacji ECMAScript 6 stanowiły syntaktyczny dodatek dla modelu klasycznego dziedziczenia w ECMAScript 5, ale później dodano do nich wiele funkcji, aby zmniejszyć poziom popełnianych błędów.

Klasy w specyfikacji ECMAScript 6 działają z dziedziczeniem prototypowym przez zdefiniowanie niestatycznych metod w prototypie klasy, natomiast metody statyczne są umieszczane w konstruktorze. Wszystkie metody nie poddają się wyliczeniu, co lepiej odpowiada zachowaniu wbudowanych obiektów, których metody domyślnie zwykle nie poddają się wyliczeniu. Ponadto konstruktor klasy nie może być wywołany bez słowa kluczowego `new`, co chroni przed przypadkowym wywołaniem klasy jako funkcji.

Dziedziczenie oparte na klasie pozwala na utworzenie klasy pochodnej na podstawie innej klasy, funkcji lub wyrażenia. Oznacza to możliwość wywołania funkcji w celu ustalenia prawidłowej bazy, po której będzie odbywać się dziedziczenie.

Podczas tworzenia nowej klasy możesz więc użyć domieszek oraz innych wzorców. Teraz dziedziczenie po wbudowanych typach danych, np. `Array`, działa zgodnie z oczekiwaniami.

Właściwość `new.target` w konstruktorze klasy może zachowywać się odmiennie, w zależności od sposobu wywołania klasy. Najczęściej możliwość tę wykorzystuje się do utworzenia abstrakcyjnej klasy bazowej, która zgłasza błąd podczas próby bezpośredniego utworzenia jej egzemplarza, choć nadal pozwala na dziedziczenie poprzez inne klasy.

Ogólnie rzecz biorąc, klasy są ważnym dodatkiem do języka JavaScript. Oferują znacznie zwięźlejszą składnię oraz lepszą funkcjonalność i pozwalają na tworzenie obiektów niestandardowych typów w bezpieczny i spójny sposób.

Skorowidz

F

G

I

K

L

Ł

M

N

O

P

R

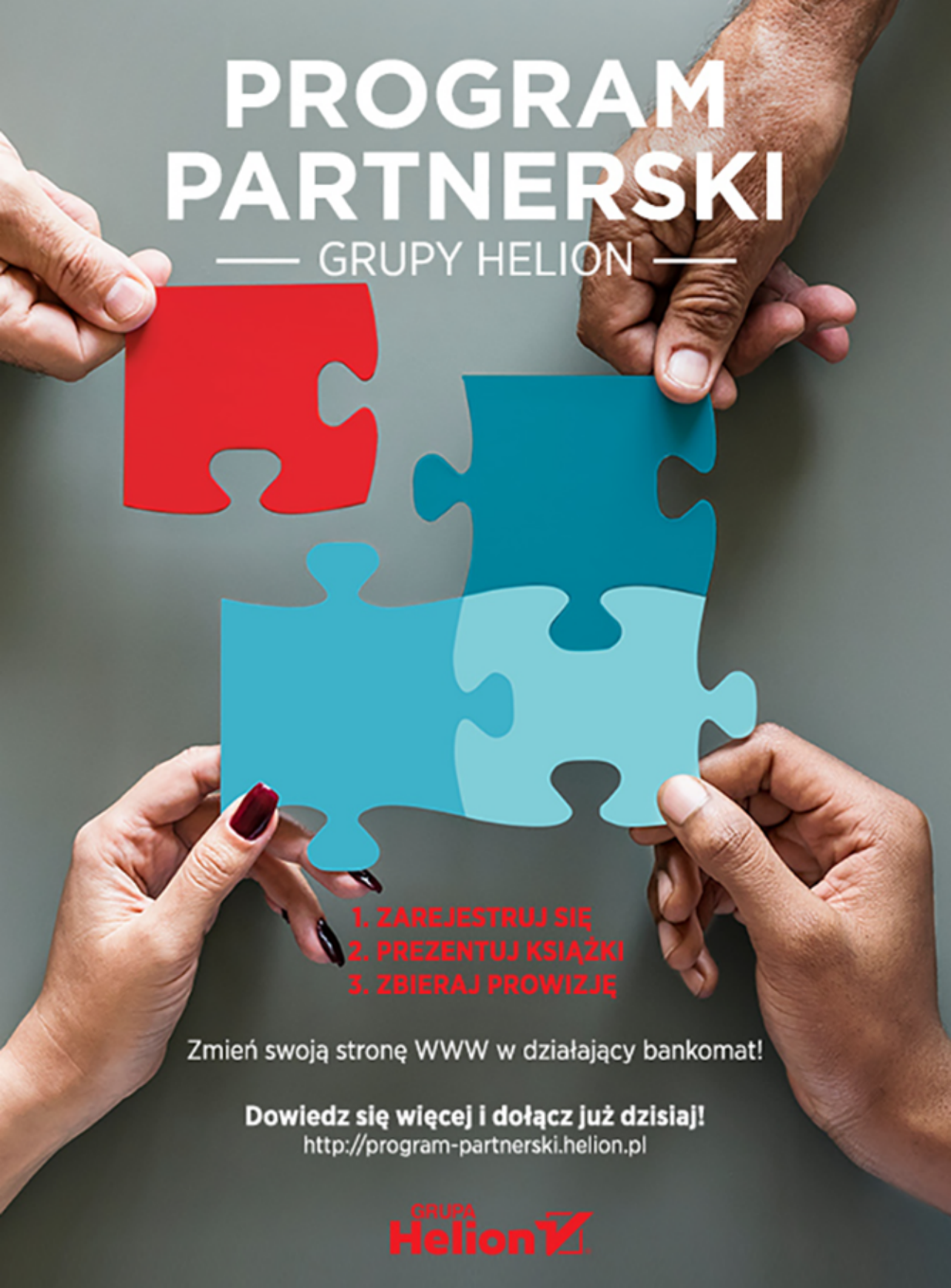
S

T

U

W

Z


PROGRAM
PARTNERSKI
GRUPY HELION
1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ
Zmień swoją stronę WWW w działający bankomat!
Dowiedz się więcej i dołącz już dzisiaj!
http://program-partnerski.helion.pl
GRUPA
Helion

JAVASCRIPT TO DOJRZAŁOŚĆ, ELASTYCZNOŚĆ I NAJNOWSZE STANDARDY!

Specyfikacja ECMAScript 6 jest najważniejszym uaktualnieniem w dotychczasowej historii języka JavaScript. Zrozumienie jej ma kluczowe znaczenie dla wszystkich programistów JavaScriptu. Tworzy ona solidny fundament i to właśnie na nim będą budowane wszystkie aplikacje JavaScriptu w przyszłości.

Oto podręcznik przeznaczony dla średnio zaawansowanych i zaawansowanych programistów JavaScriptu, którzy korzystają ze środowiska przeglądarki WWW lub Node.js. Omówiono tu zagadnienia wiązania bloków, ciągów tekstowych, wyrażeń regularnych, a także zmiany wprowadzone w funkcjach. Przedstawiono pełne wprowadzenie do typów obiektów oraz składni, które pojawiły się w JavaScripcie wraz ze specyfikacją ECMAScript 6. Nie zabrakło przykładów kodu działającego w dowolnym środowisku JavaScriptu. Dodatkowo zaprezentowano zmiany wprowadzone wraz z najnowszym standardem ECMAScript 7 (2016).

W książce omówiono między innymi:

- natywne tablice JavaScriptu i ich nowe możliwości
- obietnice i programowanie asynchroniczne
- API refleksji
- wykorzystanie proxy do kontroli obiektów
- hermetyzację kodu za pomocą modułów

Nicholas C. Zakas — pisze aplikacje internetowe od niemal dwudziestu lat. Jest doskonale znanym i uznanym ekspertem w dziedzinie tworzenia front-endu; przyczynia się do kształtowania najlepszych praktyk w tym zakresie. Specjalizuje się w stosowaniu takich technik jak JavaScript, Dynamic HTML, CSS, XML oraz XSLT. Kilka lat pracował w firmie Yahoo!, w której pełnił funkcję naczelnego inżyniera do spraw związanych z jej główną witryną. Jest autorem wielu książek dotyczących technik programistycznych.

ISBN 978-83-283-3403-8
9 788328 334038
cena: 67,00 zł

no starch press